DETAILED DESCRIPTION

[0008] Figure 1 of the drawings shows an optimizing compiler 10 in which embodiments of the invention may be practiced. The optimizing compiler 10 includes a lexical analyzer 12 which takes a source program and breaks it up to meaningful units called tokens. A syntax analyzer 14 determines the structure of the program and of the individual statements therein by grouping the tokens into grammatical phrases which are then checked by a semantic analyzer 16 for semantic errors. The compiler 10 further includes an intermediate code generator 18 which generates an intermediate program representation of the source program in an intermediate language. A code optimizer 20 attempts to optimizer program representation. The final phase of the compiler 10 is carried out by a code generator 22 which generates target comprising machine or assembly code.

[0009] In determining what optimizations to make, the code optimizer 20 performs a value analysis of the intermediate language program. Examples of such analysis include constant propagation, range analysis of subscript values, and type inference in dynamically typed programs.

[0010] The present invention permits value analysis problems to be solved over large input programs without excessive time or space penalties. In particular, program analysis according to embodiments of the invention, includes constructing dependent flow graphs in which the number of edges in the dependence flow graph is linear in the number of nodes in the graph, i.e. there are a constant number of edges per node. Embodiments of the invention make

use of an equivalence class based alias analysis of the intermediate language program to create dependence flow graphs which have the property that the edge cardinality is independent of the definition-use structure of the program being analyzed. An equivalence class is a class of overlapping memory accesses.

[0011] For purposes of describing the present invention, it is assumed that assignment statements in the intermediate language have the following syntax:

```
E :   (PUT V E)
  |   (INTEGER Z)
  |   (ADD E E)
  |   (SUB E E)
  |   (GET V)

V :   variable
Z :   integer
```

[0012] It is assumed further that INTEGER, ADD, SUB, and GET expressions all have the same type; the exact nature of the type (e.g., how many bits) is irrelevant. An assignment statement must be a PUT expression, and a PUT expression cannot be the subexpression of another subexpression.

[0013] (PUT V E): This statement writes a value to a variable. The expression E gives the value which is written to the location. V specifies a variable. It is assumed that variables are named by integers, and that other than to distinguish one variable from another, these integer names have no significance. It is also assumed that there is no aliasing or overlap among the variables used in PUT and GET expressions.

(INTEGER Z): This is the expression for an integer constant. Z is an integer literal that gives the value of the constant.

(ADD E E): This expression returns the sum of its arguments.

(SUB E E): This expression returns the difference of its arguments.

(GET V): This expression reads from the variable named by V and returns its value.

[0014] Only the syntax for the assignment statements of the intermediate language (the PUT expressions) have been shown in the above example. The reason for this is that only the PUT expressions are necessary to describe flow-insensitive program analysis in accordance with the present invention. However, it will be appreciated that a realistic intermediate language will include control flow constructs and other operators, not necessary for the present description.

[0015] **Figure 2** of the drawings shows a dependence flow graph constructed in accordance with one embodiment of the invention. In constructing the dependence flow graph shown in **Figure 2**, the PUT and GET expressions in the program are labeled with an alias. An alias, as used herein, is an equivalence class of PUT and GET expressions. An equivalence relation over aliases has the property that if there is a program execution in which two PUT and/or GET expressions in the program access the same storage location during that execution, then the two PUT and/or GET expressions have the same alias number. In other words, the equivalence relation over aliases summarizes the dependence structure of the program. Any alias analysis technique that